C:\Users\prohorenkosv.NESK.000\AppData\Local\Programs\Python\Python310-32\Scripts  
C:\Users\prohorenkosv.NESK.000\build\_env\Scripts

pip install auto-py-to-exe – для создания exe  
pip install pyodbc – для подключения к sql server  
pip install --upgrade pip  
pip install pyinstaller --upgrade  
pip install openpyxl – для работы с экселем  
pip install pyqt5 – для создания интерфейса  
pip install opencv-python – обработка полученного изображения  
pip install Pillow pyscreenshot – создание скринов экрана  
pip install PyAutoGUI – работа с мышью и клавой командами для автоматизации.  
pip install pdf2docx – преобразование из пдф в ворд(при ошибке надо закоментировать 279 строку в RawPage.py)  
pip install pycryptodome – для криптопро?  
pip install configparser

pip install django

[prohorenkosv@nesk.ru](mailto:prohorenkosv@nesk.ru)

Ctrl+/ - массово закомментировать или раскомментировать

Создание виртуальной сборки

В cmd выполните их для создания виртуальной среды

python -m venv build\_env

cd build\_env

переходим в путь\build\_env\Scripts

Устанавливаем те библиотеки, которые нужны конкретно в нужном проекте:

pip install pyinstaller

Exe, созданный с использованием виртуальной среды, будет быстрее и меньше по размеру за счет исключения ненужных библиотек.

Команда **Print() имеет необязательные параметры:**   
1) **sep** ((separator, разделитель)) нужен как разделитель между аргументами:  
print('a', 'b', 'c', sep='\*') # a\*b\*c  
2) **end** (окончание) нужен как специальное окончание:  
print('a', 'b', 'c', end='@') # a b c@

3) пример использования **sep** и **end**:   
print('a', 'b', 'c', sep='\*', end='finish') # a\*b\*cfinish.

**Примечание:** sep = '\n' перевод после каждого аргумента на новую строку. end='\n' перевод на новую строку.

**Функции**:

**min()** – определяет минимальное значение

**max()** - определяет максимальное значение

**abs()** - находит абсолютную величину. Отрицательное преобразовывает в положительное.

**Map()** – преобразовывает каждый элемент в указанный тип данных:  
n = int(input())  
print(sum(map(int, str(n))))

**Преобразование типа данных:**

**Числовое преобразование типа:**

**int()** – преобразует строку к целому числу:   
s = '1992'  
year = int(s)

**Примечание:** минус используется для обозначения отрицательных чисел и как операция.  
num1 = -6 # унарный минус  
num2 = 17 - 7 # бинарный минус

Явно указанное численное значение в коде программы называется **целочисленным литералом**:  
n = 17 # целочисленный литерал

Дробные (вещественные) числа в информатике называют **числами с плавающей точкой**:  
num = **float**(input())

**Неявное преобразование.** Любое целое число (тип int) можно использовать там, где ожидается число с плавающей точкой (тип float), поскольку при необходимости Python автоматически преобразует целые числа в числа с плавающей точкой.

**Явное преобразование.** Число с плавающей точкой нельзя неявно преобразовать в целое число. Для такого преобразования необходимо использовать явное преобразование с помощью команды int().

Преобразование чисел с плавающей точкой в целое производится с округлением в сторону нуля, то есть int(1.7) = 1, int(-1.7) = -1

**Дополнительные операции:**

**%** - **остаток** от **деления**: 10 % 3 = 1  
**//** - **целочисленное деление**.   
Примеры:  
print(10//3) = 3. # округление в меньшую сторону  
print(-10 // 3) = -4 # округление в меньшую сторону

a = -10//3 =-4 # округление в меньшую сторону  
print(-a) = 4

**Примечание:** при делении на остаток, когда первое число меньше второго 3%10 = 3.

**Пример:** нахождение цифр числа  
num = 754  
a = num % 10 # остаток на 10 = последнему числу  
b = (num % 100) // 10 # остаток на 100 и делению на 10 = предпоследнему числу  
c = num // 100 # деление на 100 = первому числу  
print(a) print(b) print(c)

**Примечание**: для удобного чтения чисел можно использовать символ подчеркивания:

num1 = 25\_000\_000

**Алгоритм получения цифр *n*-значного числа num:**

Последняя цифра: (num % 101) // 100;

* Предпоследняя цифра: (num % 102) // 101;
* Предпредпоследняя цифра: (num % 103) // 102;
* .....
* Вторая цифра: (num % 10n-1) // 10n-2;
* Первая цифра: (num % 10n) // 10n-1.

**Цепочки сравнения:**

Код, проверяющий равенство трех переменных:

if a == b == c:

print('числа равны')

else:

print('числа не равны')

**Приоритеты логических операторов:**

* в первую очередь выполняется логическое отрицание not;
* далее выполняется логическое умножение and;
* далее выполняется логическое сложение or.

Для **явного указания порядка** выполнения условных операторов **используют скобки**.

**Cтроковый тип данных:**

**str()** – преобразует целое число в строку:   
num = 17   
s = str(17)

**len()** – считает длину строки включая все символы:  
s1 = 'abcdef'   
length1 = len(s1) # считаем длину строки из переменной s1   
length2 = len('Python rocks!') # считаем длину строкового литерала

Строки можно складывать. Данная операция называется **конкатенация** или **сцепление:**s1 = 'ab' + 'bc' # итог = abbc

Строки можно умножать на число. Такое действие повторяет строку указанное кол-во раз:  
s = 'Hi' \* 4 # итог = HiHiHiHi

**Примечание:** Тройные кавычки в Python используются для многострочного (multiline) текста:

text = '''Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python design philosophy emphasizes code readability with its notable use of significant whitespace.'''

**Индексация**

Для обращения к определенному символу в строке используются [], в которых указывается номер символа. Первый индекс [0], последний можно найти используя [-1].  
Если нам нужен список строки с индексами мы используем len(строка):  
s = 'abcdef'   
for i in range(len(s)):   
 print(s[i])

Если нужны просто значения то:  
s = 'abcdef'   
for c in s:   
 print(c)

**Срезы**s = 'abcdefghij'

|  |  |  |
| --- | --- | --- |
| [2:5] | cde | строка состоящая из символов с индексами 2, 3, 4 |
| s[:5] | abcde | первые пять символов строки |
| s[5:] | fghij | строка состоящая из символов с индексами от 5 до конца |
| s[-2:] | ij | последние два символа строки |
| s[:] | abcdefghij | вся строка целиком |
| s[1:7:2] | bdf | строка состоящая из каждого второго символа с индексами от 1 до 6 |
| s[::-1] | jihgfedcba | строка в обратном порядке, так как шаг отрицательный |
| s[::-2] | jhfdb | Строка с конца с шагом 2 |

**Примечание:** для замены какого либо индекса мы делаем два среза и используем **конкатенацию**:  
s = s[:4] + 'X' + s[5:]

**Методы строк**

**Делятся на 3 типа:**

1. **Конвертация регистра**

Метод **capitalize()** возвращает копию строки s, в которой первый символ имеет верхний регистр, а все остальные символы имеют нижний регистр**:**s = 'foO BaR BAZ quX'   
print(s.capitalize()) # итог = Foo bar baz qux

**Примечание:** Символы, не являющиеся буквами алфавита, остаются неизменными.

Метод **swapcase()** возвращает копию строки s, в которой все символы, имеющие верхний регистр, преобразуются в символы нижнего регистра и наоборот:  
s = 'FOO Bar 123 baz qUX'  
print(s.swapcase()) # итог = foo bAR 123 BAZ Qux

Метод **title()** возвращает копию строки s, в которой первый символ каждого слова переводится в верхний регистр:  
s = 'the sun also rises'  
print(s.title()) # итог = The Sun Also Rises

**Примечание:** команда не пытается различить важные и неважные слова и не обрабатывает аббревиатуры и апострофы.

Метод **lower()** возвращает копию строки s, в которой все символы имеют нижний регистр:  
s = 'FOO Bar 123 baz qUX'  
print(s.lower()) # итог = foo bar 123 baz qux

Метод **upper()** возвращает копию строки s, в которой все символы имеют верхний регистр:  
s = 'FOO Bar 123 baz qUX'  
print(s.upper()) # итог = FOO BAR 123 BAZ QUX

**2) Поиск и замена**

Каждый метод в этой группе поддерживает необязательные аргументы <start> и <end>. <start> начало строки и <end> конец.  
  
Метод **count(<sub>, <start>, <end>)** считает количество **непересекающихся** вхождений подстроки <sub> в исходную строку s:  
s = 'foo goo moo'  
print(s.count('oo')) # итог = 3  
print(s.count('oo', 0, 8)) # подсчет с 0 по 7 символ # итог = 2

Метод **startswith(<suffix>, <start>, <end>)** определяет **начинается**ли исходная строка s подстрокой <suffix>. Если исходная строка начинается с подстроки <suffix>,метод возвращает значение True, а если нет, то  False:  
s = 'foobar'  
print(s.startswith('foo')) # итог = True  
print(s.startswith('baz')) # итог = False

Метод **endswith(<suffix>, <start>, <end>)** определяет **оканчивается**ли исходная строка s подстрокой <suffix>. Метод возвращает значение True если исходная строка оканчивается на подстроку <suffix> и False в противном случае:  
s = 'foobar'  
print(s.endswith('bar')) # итог = True  
print(s.endswith('baz')) # итог = False

Метод **find(<sub>, <start>, <end>)** находит **индекс первого вхождения** подстроки <sub> в исходной строке s. Если строка s не содержит подстроки <sub>, то метод возвращает значение -1. Мы можем использовать данный метод наравне с оператором in для проверки: содержит ли заданная строка некоторую подстроку или нет:  
s = 'foo bar foo baz foo qux'  
print(s.find('foo')) # итог = 0  
print(s.find('bar')) # итог = 4  
print(s.find('qu')) # итог = 20  
print(s.find('python')) # итог = -1

Метод **rfind(<sub>, <start>, <end>)** идентичен методу find(<sub>, <start>, <end>), за тем исключением, что он ищет первое вхождение подстроки <sub> начиная с конца строки s.

Метод **index(<sub>, <start>, <end>)** идентичен методу find(<sub>, <start>, <end>), за тем исключением, что он **вызывает ошибку**  ValueError: substring not found во время выполнения программы, если подстрока <sub> не найдена.

Метод **rindex(<sub>, <start>, <end>)** идентичен методу index(<sub>, <start>, <end>), за тем исключением, что он ищет первое вхождение подстроки <sub> начиная с конца строки s.

**Примечание:** методы **find() и rfind()** являются более безопасными чем index() и rindex(), так как не приводят к возникновению ошибки во время выполнения программы

Метод **strip()** возвращает копию строки s у которой удалены все пробелы стоящие **в начале и конце** строки:  
s = ' foo bar foo baz foo qux '  
print(s.strip()) # итог = foo bar foo baz foo qux

Метод **lstrip()** возвращает копию строки s у которой удалены все пробелы стоящие **в начале** строки:  
s = ' foo bar foo baz foo qux '  
print(s.lstrip()) # итог = foo bar foo baz foo qux⎵ ⎵ ⎵ ⎵ ⎵ ⎵

Метод **rstrip()** возвращает копию строки s у которой удалены все пробелы стоящие **в конце** строки:  
s = ' foo bar foo baz foo qux '  
print(s.rstrip()) # итог = ⎵ ⎵ ⎵ ⎵ ⎵ ⎵foo bar foo baz foo qux

Метод **replace(<old>, <new>)** возвращает копию s **со всеми** вхождениями подстроки <old>, замененными на <new>:  
s = 'foo bar foo baz foo qux'  
print(s.replace('foo', 'grault')) # итог = grault bar grault baz grault qux

**Примечание:** метод **replace()** может принимать опциональный третий аргумент <count>,  который определяет количество замен:  
s = 'foo bar foo baz foo qux'  
print(s.replace('foo', 'grault', 2)) # итог = grault bar grault baz foo qux

1. **Классификация символов**

Методы в этой группе классифицируют строку на основе содержащихся в ней символов.

Метод **isalnum()** определяет, состоит ли исходная строка из буквенно-цифровых символов. Метод возвращает значение True если исходная строка является непустой и состоит **только** из **буквенно-цифровых** символов и False в противном случае:  
s1 = 'abc123'  
s2 = 'abc$\*123'  
s3 = ''  
print(s1.isalnum()) # итог = True  
print(s2.isalnum()) # итог = False  
print(s3.isalnum()) # итог = False

Метод **isalpha()** определяет, состоит ли исходная строка из **буквенных** символов. Метод возвращает значение True если исходная строка является непустой и состоит **только** из **буквенных** символов и False в противном случае:  
s1 = 'ABCabc'  
s2 = 'abc123'  
s3 = ''  
print(s1.isalpha()) # итог = True  
print(s2.isalpha()) # итог = False  
print(s3.isalpha()) # итог = False

Метод **isdigit()** определяет, состоит ли исходная строка **только** из **цифровых** символов. Метод возвращает значение True если исходная строка является непустой и состоит **только** из **цифровых** символов и False в противном случае:  
s1 = '1234567'  
s2 = 'abc123'  
s3 = ''  
print(s1.isdigit()) # итог = True  
print(s2.isdigit()) # итог = False  
print(s3.isdigit()) # итог = False

Метод **islower()** определяет, являются ли **все** **буквенные** символы исходной строки строчными (имеют нижний регистр). Метод возвращает значение True если **все буквенные** символы исходной строки являются **строчными** и False в противном случае. **Все неалфавитные символы игнорируются!**s1 = 'abc'  
s2 = 'abc1$d'  
s3 = 'Abc1$D'  
print(s1.islower()) # итог = True  
print(s2.islower()) # итог = True  
print(s3.islower()) # итог = False

Метод **isupper()** определяет, являются ли **все** **буквенные** символы исходной строки **заглавными** (имеют верхний регистр). Метод возвращает значение True если **все буквенные** символы исходной строки являются **заглавными** и False в противном случае. **Все неалфавитные символы игнорируются!**s1 = 'ABC'  
s2 = 'ABC1$D'  
s3 = 'Abc1$D'  
print(s1.isupper()) # итог = True  
print(s2.isupper()) # итог = True  
print(s3.isupper()) # итог = False

Метод **isspace()**определяет, состоит ли исходная строка **только** из **пробельных** символов. Метод возвращает значение True если строка состоит **только** из **пробельных** символов и False в противном случае:  
s1 = ' '  
s2 = 'abc1$d'  
print(s1.isspace()) # итог = True  
print(s2.isspace()) # итог = False

**Форматирование строк**

Хранить строки в переменных удобно, но часто бывает необходимо **собирать строки** из других объектов (строк, чисел и т.д.) и выполнять с ними нужные манипуляции. Для этой цели можно воспользоваться механизмом **форматирования строк**.

Код приведет к ошибке из-за числового формата переменной age:  
age = 27   
txt = 'My name is Timur, I am ' + age

В данном случе надо впосрользоваться функцией **str()**txt = 'My name is Timur, I am ' + str(age)

в Python предпочтительным способом форматирования считается использование метода **format**:  
txt = 'My name is Timur, I am {}'.format(age)

Мы передаем необходимые параметры методу **format**, а Python форматирует указанную строку и помещает их в строку на место заполнителей **{}:**age = 27   
name = 'Timur'   
profession = 'math teacher'   
txt = 'My name is {}, I am {}, I work as a {}'.format(name, age, profession)

Для наглядности и гибкости форматирования мы можем использовать порядковый номер в заполнителе: {0}, {1}, {2},.... Такой номер определяет позицию параметра, переданного методу **format**(нумерация начинается с нуля):  
txt = 'My name is {0}, I am {1}, I work as a {2}'.format(name, age, profession)

**f-строки**

В Python 3.6 появилась новая разновидность строк — так называемые **f-строки**. Если поставить перед строкой префикс f, в заполнители можно будет включить код, например имя переменной. Предыдущий код можно записать в виде:  
first\_name = 'Timur'  
last\_name = 'Guev'  
age = 27  
profession = 'math teacher'  
affiliation = 'BeeGeek'  
print(**f**'Hello, {first\_name} {last\_name}. You are {age}. You are a {profession}. You were a member of {affiliation}')

На место заполнителя {first\_name} встает значение переменной first\_name, на место  заполнителя {last\_name} встает значение переменной last\_name и т.д.

**представление строк в памяти компьютера, ASCII и Unicode**

Функция **ord** позволяет определить код некоторого символа в таблице символов Unicode. Аргументом данной функции является одиночный символ:  
num1 = ord('A') # итог = 65 (код в Unicode)num2 = ord('B') # итог = 66num3 = ord('a') # итог = 97print(num1, num2, num3)

**Примечание:** функция **ord** принимает **одиночный символ**.

Функция **chr** позволяет определить по коду символа сам символ. Аргументом данной функции является **численный** код:  
chr1 = chr(65) # итог = A  
hr2 = chr(75) # итог = K  
chr3 = chr(110) # итог = n  
print(chr1, chr2, chr3)

Функции **ord** и **chr** часто работают в паре. Мы можем использовать следующий код для вывода всех заглавных букв английского алфавита:  
for i in range(26):  
print(chr(ord('A') + i))

**Примечание:** функции **ord** и **chr** являются **взаимнообратными**. Для них выполнены равенства:  
chr(ord('A')) = 'A', ord(chr(65)) = 65.

**Списки**

Например, классическая задача [сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8) (упорядочения) некоторой последовательности требует сохранения всех данных в памяти компьютера. Увы, не сохранив, их невозможно отсортировать. И тут на помощь приходит структура данных, которая в большинстве языков программирования называется [массивом](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D1%81%D0%B8%D0%B2_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)). В Python она называется **списком**.

Структура данных (data structure) — программная единица, позволяющая **хранить и обрабатывать** множество однотипных и/или логически связанных данных.

Список представляет собой последовательность элементов, пронумерованных от 0, как символы в строке.

Чтобы создать список, нужно перечислить его элементы через запятую в квадратных скобках:  
numbers = [2, 4, 6, 8, 10] # 5 элементов  
languages = ['Python', 'C#', 'C++', 'Java'] # 4 элемента

**Примечание:** Значения, заключенные в квадратные скобки и отделенные запятыми, называются **элементами списка**.

Список может содержать значения **разных типов данных**:  
info = ['Timur', 1992, 61.5] # строковое значение, целое число и с плавающей точкой.

Следующие две строки кода создают пустой список:  
mylist = [] # пустой список  
mylist = list() # пустой список

Python имеет встроенную функцию **list(),** которая помимо создания пустого списка может преобразовывать некоторые типы объектов в списки:  
even\_numbers = list(range(0, 10, 2)) # список содержит четные числа 0,2,4,6,8  
odd\_numbers = list(range(1, 10, 2)) # список содержит нечетные числа 1,3,5,7,9

**Длиной списка** называется количество его элементов. Для того, чтобы посчитать длину списка мы используем встроенную функцию **len()** (от слова length – длина):  
numbers = [2, 4, 6, 8, 10]   
print(len(numbers)) # выводим длину списка numbers # итог = 5

Оператор in позволяет проверить, содержит ли список некоторый элемент:  
numbers = [2, 4, 6, 8, 10]  
if 2 in numbers: # проверяем содержится ли 2  
 print('Список numbers содержит число 2')  
else:  
 print('Список numbers не содержит число 2')

**Индексация и срезы аналогичны строкам.**

Для изменения диапазона листа мы можем воспользоваться срезом:

fruits = ['apple', 'apricot', 'banana', 'cherry', 'kiwi', 'lemon', 'mango'] fruits[2:5] = ['банан', 'вишня', 'киви']   
print(fruits) # итог = ['apple', 'apricot', 'банан', 'вишня', 'киви', 'lemon', 'mango']

Операция **конкатенации + и умножения на число \* аналогичны строкам.**

Встроенная функция **sum()** принимает в качестве параметра список чисел и вычисляет сумму его элементов:numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]print('Сумма всех элементов списка =', sum(numbers))

Встроенные функции **min()** и **max()** принимают в качестве параметра список и находят минимальный и максимальный элементы соответственно:numbers = [3, 4, 10, 3333, 12, -7, -5, 4]print('Минимальный элемент =', min(numbers))print('Максимальный элемент =', max(numbers))

**Отличие списков от строк**

Несмотря на всю схожесть списков и строк, есть одно очень важное отличие:**строки** — **неизменяемые**объекты, а **списки** – **изменяемые**:  
s = 'abcdefg'  
s[1] = 'x' # пытаемся изменить 2 символ (по индексу 1) строки   
приводит к ошибке:  
object does not support item assignment

Следующий программный код:  
numbers = [1, 2, 3, 4, 5, 6, 7]  
numbers[1] = 101 # изменяем 2 элемент (по индексу 1) списка  
print(numbers) # итог = [1, 101, 3, 4, 5, 6, 7]

**Примечание:** изменять отдельные символы строк нельзя, однако можно изменять отдельные элементы списков. Для этого используем индексатор и оператор присваивания.

**Добавление элементов**

Метод **append()** - **добавляет новый элемент** **в конец списка:**numbers = [1, 1, 2, 3, 5, 8, 13] # создаем список   
numbers.append(21) # добавляем число 21 в конец списка   
numbers.append(34) # добавляем число 34 в конец списка   
print(numbers) # итог = [1, 1, 2, 3, 5, 8, 13, 21, 34]

**Примечание:** список должен быть создан!

Метод **extend() - расширяет один список, добавляя к нему элементы другого списка**:  
numbers = [0, 2, 4, 6, 8, 10]   
odds = [1, 3, 5, 7]   
numbers.extend(odds)   
print(numbers) # итог = [0, 2, 4, 6, 8, 10, 1, 3, 5, 7]

**Отличие** **между** методами **append()** и **extend()** проявляется **при добавлении строки к списку**:  
words1 = ['iq option', 'stepik', 'beegeek']   
words2 = ['iq option', 'stepik', 'beegeek']   
words1.append('python')   
words2.extend('python')   
print(words1) # ['iq option', 'stepik', 'beegeek', 'python']  
print(words2) #['iq option', 'stepik', 'beegeek', 'p', 'y', 't', 'h', 'o', 'n']

**Удаление элементов**

**del** - удаляет элементы списка по определенному **индексу**:  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]   
del numbers[5] # удаляем элемент имеющий индекс 5   
print(numbers) # bnju = [1, 2, 3, 4, 5, 7, 8, 9]

Оператор **del** работает и со срезами: мы можем удалить целый диапазон элементов списка:  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]   
del numbers[2:7] # удаляем элементы с 2 по 6 включительно   
print(numbers) # итог = [1, 2, 8, 9]

Можно удалить все элементы на четных позициях (0, 2, 4, ...) исходного списка:  
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]   
del numbers[::2]   
print(numbers) # итог = [2, 4, 6, 8]

**Вывод с помощью распаковки списка**

**Вариант 1.** Вывод элементов списка через один символ пробела:  
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print(\*numbers) # итог = 0 1 2 3 4 5 6 7 8 9 10

**Вариант 2.** Вывод элементов списка, каждого на отдельной строке:  
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print(\*numbers, sep='\n')

**Примечание:** Поскольку строки содержат символы, подобно тому, как списки содержат элементы, то мы можем использовать распаковку строк точно так же, как и распаковку списков:  
s = 'Python'   
print(\*s)   
print()   
print(\*s, sep='\n')

**строковые методы**

Метод **split()** разбивает строку на слова, используя в качестве разделителя последовательность пробельных символов:  
s = 'Python is the most powerful language'   
words = s.split()   
print(words) # ['Python', 'is', 'the', 'most', 'powerful', 'language']

**Примечание:**   
numbers = input().split()

Если при запуске этой программы ввести строку 1 2 3 4 5, то список numbers будет следующим  ['1', '2', '3', '4', '5']. Обратите внимание, что список будет состоять из строк, а не из чисел. Если требуется получить именно список чисел, то затем нужно элементы списка по одному **преобразовать в числа**:  
numbers = input().split()  
for i in range(len(numbers)):   
 numbers[i] = int(numbers[i])

У метода split() есть необязательный параметр, который определяет, какой набор символов будет использоваться в качестве разделителя между элементами списка:  
ip = '192.168.1.24'   
numbers = ip.split('.') # указываем явно разделитель   
print(numbers) # ['192', '168', '1', '24']

Метод **join()** собирает строку из элементов списка, используя в качестве разделителя строку, к которой применяется метод:  
words = ['Python', 'is', 'the', 'most', 'powerful', 'language']   
s = ' '.join(words)   
print(s) # Python is the most powerful language

**Примечание:** Строковый метод **split()** служит для преобразования строки в список, а метод **join()** — для преобразования списка в строку.  
**Примечание 2:** Строковый метод **join()** работает **только со списком строк**. Следующий код приводит к ошибке:  
numbers = [1, 2, 3, 4] # список чисел  
s = '\*'.join(numbers)  
print(s)

**методы списков**

Метод **insert() позволяет вставлять значение в список в заданной позиции** (передается два аргумента): **index**: индекс, задающий место вставки значения; **value**: значение, которое требуется вставить.

Когда значение вставляется в список, список расширяется в размере, чтобы разместить новое значение. Значение, которое ранее находилось в заданной индексной позиции, и все элементы после него сдвигаются на одну позицию к концу списка:

names = ['Gvido', 'Roman' , 'Timur']  
print(names) # ['Gvido', 'Roman' , 'Timur']  
names.insert(0, 'Anders')  
print(names) # ['Anders', 'Gvido', 'Roman' , 'Timur']  
names.insert(3, 'Josef')  
print(names) # ['Anders', 'Gvido', 'Roman' , 'Josef', 'Timur']

**Примечание:** если указан недопустимый индекс, то во время выполнения программы не происходит ошибки. Если задан индекс за пределами конца списка, то значение будет добавлено в конец списка. Если применен отрицательный индекс, который указывает на недопустимую позицию, то значение будет вставлено в начало списка.

Метод **index()** **возвращает индекс первого элемента, значение которого равняется переданному в метод значению** (передается один параметр):  
**value**: значение, индекс которого требуется найти:  
names = ['Gvido', 'Roman' , 'Timur']  
position = names.index('Timur')  
print(position) # 2

Если элемент в списке не найден, то во время выполнения происходит ошибка:  
names = ['Gvido', 'Roman' , 'Timur']  
position = names.index('Anders')  
print(position)

приводит к ошибке:  
ValueError: 'Anders' is not in list

**Примечание:** чтобы избежать таких ошибок, можно использовать метод **index()** вместе с оператором принадлежности **in**:  
names = ['Gvido', 'Roman' , 'Timur']  
if 'Anders' in names:  
 position = names.index('Anders')  
 print(position)  
else:  
 print('Такого значения нет в списке')

Метод **remove()** **удаляет первый элемент, значение которого равняется переданному в метод значению** (передается один параметр):  
**value**: значение, которое требуется удалить.

Метод уменьшает размер списка на один элемент. Все элементы после удаленного элемента смещаются на одну позицию к началу списка. Если элемент в списке не найден, то во время выполнения происходит ошибка:  
food = ['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']  
print(food) # ['Рис', 'Курица', 'Рыба', 'Брокколи', 'Рис']  
food.remove('Рис')  
print(food) # ['Курица', 'Рыба', 'Брокколи', 'Рис']

**Важно:** метод **remove()** удаляет только первый элемент с указанным значением. Все последующие его вхождения остаются в списке. Чтобы удалить все вхождения нужно использовать цикл **while** вместе с оператором **in** и методом **remove**.

Метод **pop()** **удаляет элемент по указанному индексу и возвращает его** (передается один **необязательный** аргумент):  
**index**: индекс элемента, который требуется удалить.

Если индекс не указан, то метод удаляет и возвращает последний элемент списка. Если список пуст или указан индекс за пределами диапазона, то во время выполнения происходит ошибка:  
names = ['Gvido', 'Roman' , 'Timur']  
item = names.pop(1)  
print(item) # Roman  
print(names) # ['Gvido', 'Timur']

Метод **count()** **возвращает количество элементов в списке, значения которых равны переданному в метод значению** (передается один параметр):  
**value**: значение, количество элементов, равных которому,  нужно посчитать.

Если значение в списке не найдено, то метод возвращает 0:  
names = ['Timur', 'Gvido', 'Roman', 'Timur', 'Anders', 'Timur']  
cnt1 = names.count('Timur')  
cnt2 = names.count('Gvido')  
cnt3 = names.count('Josef')  
print(cnt1) # 3  
print(cnt2) # 1  
print(cnt3) # 0

Метод **reverse() инвертирует порядок следования значений в списке** (меняет его на противоположный):  
names = ['Gvido', 'Roman' , 'Timur']  
names.reverse()  
print(names) # ['Timur', 'Roman', 'Gvido']

**Важно:** существует большая разница между вызовом метода **names.reverse()** и использованием среза **names[::-1]**. Метод **reverse()** **меняет** порядок элементов на обратный **в текущем списке**, а **срез создает копию** списка, в котором элементы следуют в обратном порядке.

Метод **clear()** **удаляет все элементы из списка**:  
names = ['Gvido', 'Roman' , 'Timur']  
names.clear()  
print(names) # []

Метод **copy()** **создает** **поверхностную копию списка**:  
names = ['Gvido', 'Roman' , 'Timur']  
names\_copy = names.copy() # создаем поверхностную копию списка names  
print(names) # ['Gvido', 'Roman', 'Timur']  
print(names\_copy) # ['Gvido', 'Roman', 'Timur']

Аналогичного результата можно достичь с помощью срезов или функции list():  
names = ['Gvido', 'Roman' , 'Timur']  
names\_copy1 = list(names)# создаем поверхностную копию с помощью функции list()  
names\_copy2 = names[:] # создаем поверхностную копию с помощью среза от начала до конца

Метод **sort()** **сортирует элементы списка по возрастанию или убыванию**:  
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]  
a.sort()  
print('Отсортированный список:', a) # Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]

**По умолчанию** метод **sort()** **сортирует список по возрастанию**. Если требуется отсортировать список **по убыванию**, необходимо **явно указать** параметр **reverse = True**:  
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]  
a.sort(reverse = True) # сортируем по убыванию  
print('Отсортированный список:', a) # Отсортированный список: [1000, 99, 45, 34, 12, 9, 8, 7, 6, 1, 0, -2, -3, -67]

**Примечание: с** помощью метода **sort()** **можно сортировать** списки содержащие не только числа, но и **строки**. В таком случае элементы списка сортируются в соответствии с [лексикографическим порядком](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D0%BA%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA):  
a = ['бета', 'альфа', 'дельта', 'гамма']  
a.sort()  
print ('Отсортированный список:', a) # Отсортированный список: ['альфа', 'бета', 'гамма', 'дельта']

**Примечание 2:** метод **sort(**) использует алгоритм [Timsort](https://ru.wikipedia.org/wiki/Timsort" \t "_blank).

**Примечание 3:** чаще требуется **сохранять исходный** список. Тогда создаем новый отсортированный с помощью **функции sorted:**  
a = [5, 3, 1, 2, 4]  
b = sorted(a)  
print(b)   #  [1, 2, 3, 4, 5]  
print(a)   #  [5, 3, 1, 2, 4]

**Списочные выражения**

В Python есть **механизм для создания списков** **из неповторяющихся элементов**. Такой механизм называется — **списочное выражение** (list comprehension):  
numbers = [i for i in range(10)]

Общий вид списочного выражения следующий:

**[выражение for переменная in последовательность]**

где **переменная** — имя некоторой переменной, **последовательность** — последовательность значений, которые она принимает (список, строка или объект, полученный при помощи функции range), **выражение** — некоторое выражение, как правило, зависящее от использованной в списочном выражении переменной, которым будут заполнены элементы списка.

**Примеры использования списочных выражений:**

**1. Создать список, заполненный 10 нулями можно и при помощи списочного выражения:**zeros = [0 for i in range(10)]

**2. Создать список, заполненный квадратами целых чисел от 0 до 9 можно так:**squares = [i \*\* 2 for i in range(10)]

**3. Создать список, заполненный кубами целых чисел от 10 до 20 можно так:**cubes = [i \*\* 3 for i in range(10, 21)]

**4. Создать список, заполненный символами строки:**chars = [c for c in 'abcdefg']  
print(chars)

**Считывание входных данных**

При решении многих задач из предыдущих уроков мы считывали начальные данные (строки, числа) и заполняли ими список. С помощью списочных выражений процесс заполнения списка можно заметно сократить.

Например, если сначала вводится число n – количество строк, а затем сами строки, то создать список можно так:  
n = int(input())  
lines = [input() for \_ in range(n)]

Можно опустить описание переменной n:  
lines = [input() for \_ in range(int(input()))]

Если требуется считать список чисел, то необходимо добавить преобразование типов:  
numbers = [int(input()) for \_ in range(int(input()))]

**Примечание:** **списочные выражения** часто **используются для инициализации списков.** В Python **не принято создавать пустые списки**, а затем заполнять их значениями, если можно этого избежать.

**Условия в списочном выражении**

В **списочных выражениях** можно **использовать условный оператор**. Например, если требуется создать список четных чисел от 0 до 20, то мы можем написать такой код:  
evens = [i for i in range(21) if i % 2 == 0]

**Важно:** для того, чтобы получить список, состоящий из четных чисел, лучше использовать функцию range(0, 21, 2). Предыдущий пример приведен для демонстрации возможности использования условий в списочных выражениях.

**Вложенные циклы**

В списочном выражении можно использовать вложенные циклы:  
numbers = [i \* j for i in range(1, 5) for j in range(2)]  
print(numbers) # [0, 1, 0, 2, 0, 3, 0, 4]

Такой код равнозначен следующему:  
numbers = []  
for i in range(1, 5):  
 for j in range(2):  
 numbers.append(i \* j)  
print(numbers)

Подводя итог:  
word = 'Hello', numbers = [1, 14, 5, 9, 12], words = ['one', 'two', 'three', 'four', 'five', 'six'].

| **Списочное выражение** | **Результирующий список** |
| --- | --- |
| [0 for i in range(10)] | [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] |
| [i \*\* 2 for i in range(1, 8)] | [1, 4, 9, 16, 25, 36, 49] |
| [i \* 10 for i in numbers] | [10, 140, 50, 90, 120] |
| [c \* 2 for c in word] | ['HH', 'ee', 'll', 'll', 'oo'] |
| [m[0] for m in words] | ['o', 't', 't', 'f', 'f', 's'] |
| [i for i in numbers if i < 10] | [1, 5, 9] |
| [m[0] for m in words if len(m) == 3] | ['o', 't', 's'] |

**Сортировка списков**

**Алгоритм сортировки** — это **алгоритм упорядочивания элементов в списке**. Алгоритмы сортировки оцениваются **по скорости выполнения** и **эффективности** использования памяти:

1. **время** — основной параметр, характеризующий быстродействие алгоритма;
2. **память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных.

Примечание: **алгоритмы сортировки**, **не потребляющие дополнительной памяти**, относят к **сортировкам на месте**.

**Основные алгоритмы сортировки**

**Медленные:**

1. **Пузырьковая сортировка (Bubble sort);**
2. **Сортировка выбором (Selection sort);**
3. **Сортировка простыми вставками (Insertion sort).**

**Быстрые:**

1. **Сортировка Шелла (Shell sort);**
2. **Быстрая сортировка (Quick sort);**
3. **Сортировка слиянием (Merge sort);**
4. **Пирамидальная сортировка (Heap sort);**
5. **Сортировка TimSort (используется в Java и Python).**

Большинство алгоритмов сортировки, в частности, указанные выше, основаны на сравнении двух элементов списка. Существуют однако алгоритмы не основанные на сравнениях. Такие алгоритмы как правило используют наперед заданные условия относительно элементов списка. Например, элементами списка являются натуральные или целые числа в некотором диапазоне, элементами являются строки и т.д.

**К алгоритмам не основанным на сравнениях можно отнести следующие:**

1. **Сортировка подсчетом (Counting sort);**
2. **Блочная сортировка (Bucket sort);**
3. **Поразрядная сортировка (Radix sort).**

**Сортировка пузырьком**

Требуется отсортировать по возрастанию список чисел:   
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99].

Следующий программный код реализует алгоритм пузырьковой сортировки:  
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]  
n = len(a)  
for i in range(n - 1):  
 for j in range(n - i - 1):  
 if a[j] > a[j + 1]: # если порядок элементов пары неправильный  
 a[j], a[j + 1] = a[j + 1], a[j] # меняем элементы пары местами print('Отсортированный список:', a)

Результатом выполнения такого кода будет:  
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]

**Примечание:** алгоритм пузырьковой сортировки можно немного ускорить. Если на одном из очередных проходов окажется, что обмены больше не нужны, то это означает, что все элементы списка находятся на своих местах, то есть список отсортирован. Для реализации такого ускорения нужно воспользоваться **сигнальной меткой**, то есть **флажком** и оператором прерывания **break**.

**Сортировка простыми вставками**

Требуется отсортировать по возрастанию список чисел:   
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99].

Следующий программный код реализует алгоритм сортировки простыми вставками:  
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]  
n = len(a)  
for i in range(1, n):   
 elem = a[i] # первый элемент из неотсортированной части списка  
 j = i  
 while j >= 1 and a[j - 1] > elem:   
 a[j] = a[j - 1]  
 j -= 1  
 a[j] = elem  
print('Отсортированный список:', a)

Результатом выполнения такого кода будет:  
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]

**Примечание:** алгоритм сортировки простыми вставками можно значительно ускорить если осуществлять поиск нужной позиции для вставки очередного элемента из неотсортированной части списка с помощью [бинарного поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA).

**Модули**

В Python **модулем** называется **библиотека функций**, которую можно подключать к своим программам. **Модуль math – один из наиважнейших в Python**. Этот модуль предоставляет обширный функционал для проведения вычислений с вещественными числами (числами с плавающей точкой).

Для использования этих функций в начале программы необходимо **подключить модуль**, что делается командой **import**:

import math

num1 = math.sqrt(2) # вычисление корня квадратного из двух   
num2 = math.ceil(3.8) # округление числа вверх   
num3 = math.floor(3.8) # округление числа вниз   
print(num1) # 1.4142135623730951  
print(num2) # 4  
print(num3) # 3

Для того, **чтобы не указывать** название модуля и символ точки при вызове функций:  
from math import \*

num1 = sqrt(2) # вычисление корня квадратного из двух   
num2 = ceil(3.8) # округление числа вверх   
num3 = floor(3.8) # округление числа вниз

**При таком способе** подключения, **импортируются абсолютно все**функции модуля math. Если нужно использовать только **некоторые функции** модуля, то мы перечисляем их после import:  
from math import sqrt, ceil

print(sqrt(25))   
print(ceil(34.7))   
print(floor(12.8)) # приведет к ошибке, так как функция floor не подключен

Список функции модуля math:

**round(x)** - Округляет число x до ближайшего целого.

**round(x, n)** - Округляет число x до n знаков после точки

**pow(x, n)** - Возведение числа x в степень n

**factorial(n)** - Факториал натурального числа n

**Циклы**

В Python существует две основных разновидности циклов:

* циклы, повторяющиеся определенное количество раз (for, счетные циклы, **counting loops**);
* циклы, повторяющиеся до наступления определенного события (while, условные циклы, **conditional loops**).

**Однократное выполнение цикла** называется **Итерацией**.

**Цикл for:**

В принципе понятный цикл:   
for i in range(5):  
 num = int(input())   
print('Квадрат вашего числа равен:', num \* num) #поиск квадрата каждого числа  
print('Цикл завершен')

**Примечание:**range(1, 100, 2)с параметрами: **первый** параметр отвечает **за начало** последовательности, **второй за конец** последовательности(**НЕ включительно**) и **третий** отвечает за **шаг последовательности**.

range(20, 16, -1) создаст обратную последовательность чисел 20, 19, 18, 17

**Цикл While:**

num = int(input())   
while num != -1: # цикл выполняется пока введенное значение не равно -1. Как только -1, так цикл прервется.  
 print('Квадрат вашего числа равен:', num \* num)   
 num = int(input()

считывание до стоп-значения:  
text = input()# первое введенное значение, которое проверяется в цикле на стоп-значение  
total = 0   
while text != 'stop':   
 num = int(text)   
 total += num   
 text = input() # новое значение, которое проверяется в цикле на стоп-значение.  
print('Сумма чисел равна', total)

определение, есть ли в числе цифра 7:  
num = int(input())   
has\_seven = False # сигнальная метка . по умолчанию думаем, что в числе нет цифры 7.  
while num != 0:   
 last\_digit = num % 10 # проверяем начиная с последней цифры  
 if last\_digit == 7:   
 has\_seven = True # если проверка положительна, то меняем метку на правду

break # прерываем если нашли искомое  
 num = num // 10 # отсекаем последнюю цифру от числа в случае несоблюдения условия  
if has\_seven == True:   
 print('YES')   
else:   
 print('NO')

Иногда цикл **надо прервать**, в этом нам помогает оператор **break**

Иногда **надо пропустить** итерацию и **перейти к следующей**, в этом помогает оператор **continue**.

for i in range(1, 101): # выводим числа от 1 до 101.  
 if i == 7 or i == 17 or i == 29 or i == 78: # кроме 7,17,29 или 78  
 continue # переходим на следующую итерацию если условие соблюдено  
print(i)

**Примечание:** Блок **else** в циклах часто применяется для обработки отсутствия элементов:  
n = int(input())   
while n != 0:   
 last = n % 10   
 if last == 7:   
 print('Число', n, 'содержит цифру 7')   
 break   
 n //= 10   
else:   
 print('Число', n, 'не содержит цифру 7')

**Обмен значений переменных**

В результате выполнения такого кода Python поменяет значения переменных x и y местами.

x, y = y, x

**Примечание:** переменных можно задать больше одновременно.

**Сигнальные метки**

**Сигнальная метка (флажок)** может использоваться, когда надо чтобы одна часть программы узнала, о происходящем в другой части программы.  
Напишем программу, определяющую, что натуральное число является [простым](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE):

num = int(input())

flag = True # задаем флаг изначально подразумевая статус, что число простое

for i in range(2, num):

if num % i == 0: # если исходное число делится на какое-либо отличное от 1 и самого себя

flag = False # устанавливаем статус ложь, если делитель нашелся

if num == 1:

print('Это единица, она не простая и не составная')

elif flag == True:

print('Число простое')

else:

print('Число составное')

django  
Установка: python -m pip install Django

Версия: python -m django –version

Создаем проект: django-admin startproject mysite

Запускаем сервер: python manage.py runserver (python manage.py runserver имя сервера/ip:8000)

Создаем приложение: python manage.py startapp polls

**python** -m **pip** uninstal Django

Создание виртуальной сборки

В cmd выполните их для создания виртуальной среды

python -m venv build\_env

cd build\_env

переходим в путь\build\_env\Scripts

Устанавливаем те библиотеки, которые нужны конкретно в нужном проекте:

pip install pyinstaller

Exe, созданный с использованием виртуальной среды, будет быстрее и меньше по размеру за счет исключения ненужных библиотек.